Week 2: Agentic Workflows & GitHub

September 2, 2026

THE LOOP THIS WHOLE COURSE RUNS ON

An **agent** is an AI that doesn't just talk back — it reads your files, writes new ones, runs commands, and reports what it did. Claude Code is the one we use. You supply the project and the direction; it does the typing. That changes who does the work, but not what makes the work *good*. The loop is:

prompt → read the diff → verify → commit / push

The pipeline underneath is the same one it has always been: a data file, a script, a figure. AI can build the first three steps. **Verifying the output is still yours.**

> An agent almost never makes a **syntax** mistake — no typos, no missing brackets, a tidy folder layout. So a wrong answer arrives looking exactly like a right one. **There is no version of a bad result that throws an error.** Running is not the same as right.

VERSION CONTROL WITH GITHUB

Git is a **version control system**: a project folder that tracks its own history. GitHub is where that history lives online so other people can see it. We drive it entirely through **GitHub Desktop** — every operation below is a button. (The command line works too, if you'd rather.)

Term	What it means
repo	A project folder that tracks its own history
commit	A labeled save point worth remembering
push	Send your commits up to GitHub
pull	Bring GitHub's commits down to your computer
clone	Make your own local copy of someone else's repo
collaborator	Someone given write access, so their pushes land in your repo
branch	A sandbox copy where you work without touching `main`

The solo loop

Make changes locally, then **commit** them with a message, then **push**. Repeat. That's the whole thing when you're working alone.

Commit messages are notes to future you. "stuff", "update", "fixed it" tell you nothing in November. "Add flights bar chart", "Fix airline join dropping NAs", "Clean column names in import" tell you everything.

What not to commit

A **`.gitignore`** file lists the things git should pretend it can't see. Keep out rendered output (`_site/`, `*_files/`), OS and editor junk (`.DS_Store`, `.Rhistory`), anything holding a password or API key, and data files that are huge or private. Commit your source (`.qmd`, `.R`), small data files, your `README.md` and `CLAUDE.md`, and the `.gitignore` itself. Ignored files still sit on your own computer — they just never get pushed.

The collaboration loop

To let someone into your repo: on GitHub.com, **Settings › Collaborators**, add their username, they accept the emailed invite.

Once you're a collaborator on a repo, you contribute the same way you work alone, plus one step at the front: **clone** it, **pull**, make your changes, **commit**, **push**. Everyone works on `main`. Pull before you start, or you'll be editing yesterday's version of the project.

If two people edit different files — or different parts of the same file — git merges the work for you. If you both change the *same lines*, git stops and asks which version you want. That's a **merge conflict**, and you resolve it by editing the file. Pulling often and committing small changes is what keeps them rare.

A **branch** is the other way to do this: a sandbox copy of `main` where you work, then open a **pull request** to merge it back. Large teams live this way, and you'll see it everywhere on GitHub. We'll stay on `main` all semester.

Publishing the site with GitHub Pages

A public repo can serve a website straight from its own files, for free:

1. Ask Claude to change the Quarto output folder from `_site` to `docs` — that's the `output-dir` key in `_quarto.yml` — then re-render.
2. **Commit and push.**
3. On GitHub.com, go to the repo's **Settings › Pages**.
4. Under **Build and deployment**, set **Source: Deploy from a branch**, **Branch: main**, **Folder: /docs**, then **Save**.
5. Wait a minute, then visit the URL GitHub shows at the top of that page.

The repo has to be **public** for this to work. Your final project report will be a repo that renders exactly this way.

CLAUDE CODE IN POSITRON

1. Make a folder — **no spaces in the name**.
2. In Positron: **File › Open Folder…** and pick that folder.
3. In the terminal pane, type `claude` and press Enter.

Claude Code is one *harness* among several. Google's Gemini CLI and OpenAI's Codex are the same idea with different plumbing.

Modes: your permission dial

Press **shift + tab** to cycle. The current mode always shows at the bottom of the prompt box.

Mode	What it does
Plan	Reads only; drafts a plan before doing big work
Manual	You approve or deny each edit
Accept Edits	Auto-applies file edits
Auto	Approves everything — full file access

Left to right is less freedom / more supervision → more freedom / less supervision. Hand over more autonomy only when you already have a very clear step-by-step plan.

SLASH COMMANDS

Command	What it does
`/init`	Writes a `CLAUDE.md` for the project
`/model`	Switch which model you're talking to
`/effort`	Set how hard the model thinks
`/compact`	Summarize the chat so far, keep going
`/clear`	Wipe the conversation, start fresh

`CLAUDE.md`: standing instructions

A plain markdown file the agent **reads at the start of every session** — your project's house rules. What this project is, how to run it, what you want done differently.

Two places it can live: a **project** `CLAUDE.md` in the repo root (rules for this repo, shared with anyone who clones it), and a **home** `~/.claude/CLAUDE.md` (rules for everything you do, on your machine only).

You don't write it from scratch. `/init` walks the folder — file tree, README, config files, existing code — works out what the project is and how it gets built, and writes the file for you.

But `/init` output is a draft, not gospel. Claude is guessing from what it can see, and it gets things wrong. Read it: is the description right? Are the build commands real? Did it invent a convention you don't follow? Then edit by hand — delete what's wrong, and add the rules it *can't* know, like "charts use `theme_minimal_hgrid()`".

`/model` vs. `/effort`

`/model` picks **which brain**. `/effort` picks **how long it thinks** before it starts editing files.

Model	Good for
Opus	Hard reasoning, tricky bugs, big refactors
Sonnet	Everyday work — fast, capable, cheaper
Haiku	Quick, simple, high-volume tasks

Effort	What it's for
`low`	Simple, mechanical edits — fastest, cheapest
`medium`	Everyday work
`high`	Multi-step tasks, debugging, design decisions
`xhigh`	Genuinely hard problems — slowest, priciest

Bigger isn't always better. Turn effort **up** when the hard part is *deciding what to do*; turn it **down** when you already know exactly what you want. `/effort auto` returns to the model's default.

TOKENS, BUDGET, AND CONTEXT

A **token** is a chunk of text, roughly three-quarters of a word. Everything the agent does is paid for in tokens: thinking before it acts, running tools (every file it reads, every command it runs), and writing code back to you.

Your budget refills on two clocks. The **5-hour** window starts at your first message and then resets — that's the one you'll actually hit. A **7-day** cap sits on top as the safety net. Run out mid-assignment and you wait.

Context is the other meter. Every message re-sends the *whole* conversation, not just your last line.

Context used	What it means
0–60%	Room to think — healthy
60–80%	Slower and pricier — trim it soon
80–100%	Hard stop at 100% — start fresh

Two consequences: a tiny question pays for the whole history, and you should start fresh **early**, not at 99%.

`/clear` wipes the slate — use it between unrelated tasks. `/compact` condenses the conversation and keeps going — use it mid-task when things get long. A good `CLAUDE.md` is what survives a `/clear`.

The agent's chattiness is billed too, so tell it to be terse. The Caveman plugin does exactly that — about 65% fewer tokens on prose, 8.5% on real coding runs, accuracy unchanged.

SKILLS

`/init`, `/clear`, and `/effort` are built into Claude Code. A **skill** is a slash command *you* define that teaches the agent one of *your* recipes: write the conventions down once, then invoke them by name forever.

Skills live in `~/.claude/skills/` (yours everywhere) or `.claude/skills/` inside a repo (shared with that project). Each is a folder with a `SKILL.md` in it, and you invoke it as `/skill-name`.

We used `/my-chart-style` — the same request ("mean departure delay by airline, as a bar chart") with and without it. Without: a gray default `geom_col()` in whatever order the rows happened to be in. With: sorted bars, a cowplot minimal theme whose gridlines match the bar direction, a white background, real axis labels, a title that states the finding, and a source caption.

Skills save time, make output reproducible, and keep it consistent across a project. Learn the conventions once, then encode them.

WORKING SAFELY

Agents are **confidently wrong**. They won't say "I'm not sure." Throughput goes up; quality is not guaranteed. Fast is not the same as better.

1. Ask for code, not results

If you ask for an answer, you get a number you can't check. If you ask for a script, you get something you can read, re-run, and hand to someone else. Keep the work as a `data → script → figure` pipeline, whoever typed it.

2. Run sanity checks

Never ask the agent whether it's right — it will say yes. Check it a way it can't fake: a **published number** (codebook, prior paper, official table), a **hand calculation** you do yourself, or a **second analyst** (different chat, different prompt, different model).

Every time, ask:

- Did **row and column counts** change the way I expected?
- Did I **read the code**, not just the summary?
- Did I **spot-check** values against the source file?
- Were rows **silently dropped** — `NA` handling, inner vs. left join?
- Do **totals and aggregates** make sense?
- Is it **reproducible**, or did it hard-code an answer?

A sanity-check script is short and boring: row count survived the merge (1,200 to 1,200), every ID appears once (0 duplicates), no percentage above 100%. Two checks pass silently; the third catches something real.

3. Use fake data

Sometimes you can't hand the data over at all — medical records, student grades, proprietary sales figures, anything under an NDA or IRB protocol. So don't. Build a **small fake table with the same columns and types**, have the agent write the script against that, then run the same script on the real data on your own machine. The agent never sees the real data; the script still works on it.

In R, the **charlatan** package generates fake names, jobs, phone numbers, emails, addresses, and coordinates — `ch_name()`, `ch_job()`, or `ch_generate()` for a whole data frame at once. Set a seed and it's reproducible like any other RNG. Python's **faker** package is the same tool; charlatan is an R port of it.

THE AIRLINE DELAY TRAP

We asked for a ranking of airlines by mean departure delay, joined to the airline names. The code ran. The chart was clean. The answer was wrong three different ways, and every one was invisible in both the chart and the code:

What broke	What actually happened
Sample size	Alaska ranked worst on 20 flights; Mesa 10; Hawaiian 11
A vanished airline	SkyWest flew once, cancelled — `mean()` gave `NaN`, and the bar silently disappeared
Cancellations	`na.rm = TRUE` deletes every cancelled flight; Endeavor cancelled 7.3%, none counted as "late"

The lesson isn't about airlines. It's that a clean chart is not evidence of a correct chart, and you only find these by going looking.

SNAGS YOU'LL HIT THIS WEEK

Symptom	Usual cause
Repo isn't in GitHub Desktop	Made on GitHub.com, never cloned — **File › Clone Repository**
`claude` isn't a known command	Not installed, or the terminal was already open when you installed it — open a new one
Agent edits files somewhere unexpected	You opened a *file* in Positron, not the *folder*
Push is rejected as out of date	Someone pushed first — pull, then push
Your partner's change isn't in your files	You never pulled — **Fetch origin**, then **Pull**
Merge conflict markers (`<<<<<<<`) in a file	You both edited the same lines — pick what you want, delete the markers, commit
Pages URL shows a 404	`output-dir` isn't `docs`, `docs/` never got pushed, or the repo is private
Agent forgets the project after `/clear`	It only remembers what's in `CLAUDE.md` — write it down there
A skill never fires	Wrong folder, or `SKILL.md` is missing its `name` / `description` header

BEFORE NEXT CLASS

- **HW2** — read forward on data wrangling, try the light practice, reflect. Due Sept 7.
- **Quiz 1** at the start of next class, covering weeks 1–2.
- Sign up for a team on the team-formation sheet and skim the project ideas. The **proposal is due Sept 20** — it starts brewing now.
- Next week: **Data Wrangling**

You should be able to:

- Create a repo, commit, and push it, all from GitHub Desktop.
- Clone a repo you've been added to, commit to `main`, push, and pull your collaborator's work back down.
- Say what belongs in a `.gitignore` and why.
- Open a folder in Positron and launch `claude` in it.
- Cycle modes with shift + tab and say what each one lets the agent do.
- Use `/init`, `/model`, `/effort`, `/compact`, and `/clear`, and edit the `CLAUDE.md` that `/init` writes.
- Install and invoke a skill, and change what it produces.
- Name three ways a clean chart can still be wrong, and check for each.

If any of those are shaky, that's what Slack and office hours are for.